LISP is a source language for writing algorithms. Unlike most other languages, in which a source program consists of a sequence of instructions to be executed in a certain order, a LISP program usually (but not always) consists of a collection of function definitions.

The basic object in LISP is a branching-tree type of structure called an S-expression. All function definitions, function arguments, function values, programs, instructions, variables, constants, and data are S-expressions. At each node of an S-expression there are two branches. The branches eventually terminate in atoms. An S-expression is thus defined recursively - it is either an atom or the concatenation of two S-expressions.

[Diagram of an S-expression]

**fig. 1 - an S-expression**

In the S-expression of figure 1, `lambda`, `x`, `nil`, `times`, and `2` are atoms. The S-expression is the concatenation of the S-expression `lambda`, which is an atom, with another concatenation.

There is a notation for transmission of S-expressions between LISP and the outside world. An atom is simply spelled out. A concatenation is written as a left parenthesis, the S-expression on the left side, a period, the S-expression on the right side, and a right parenthesis. The S-expression of figure 1 could be written

\[
(l\{\text{lambda}\}.(\{x,\text{nil}\}).(\{\text{times},(\{2,\text{x},\text{nil}\})\}).\text{nil}))
\]

Almost all S-expressions that are commonly used have a chain of branches going off to the right, with the atom `nil` at the end. From each node an S-expression hangs on the left. Such an S-expression is called a list.
fig. 2 - a list

There is a special notation for lists, consisting of a left parenthesis, the items of the list with spaces separating them, and a right parenthesis. Using list notation, the S-expression of figure 2 is written

\[(a \ b \ x \ y \ z)\]

and that of figure 1 is written

\[(\text{lambda } (x) \ (\text{times } 2 \ x \ x))\]

Either format is permissible for input. In typing out S-expressions, LISP uses list notation wherever possible.

There are two types of atoms. Numbers consist of one or more digits, with or without a minus sign. Atomic symbols contain at least one non-numeric character.

The three most basic functions in LISP are \texttt{car}, \texttt{cdr}, and \texttt{cons}. These three, along with over 50 others, exist as subroutines in the LISP program.

Given a nonatomic S-expression as an argument \texttt{car} finds the left half. \texttt{Cdr} finds the right half.

\[
\text{car of } (a.b) = a \\
\text{cdr of } (a.b) = b
\]

\texttt{Cons} takes two arguments and concatenates them.

\[
\text{cons of } a \text{ and } b = (a.b)
\]

Note that the \texttt{car} of a list is its first element, and the \texttt{cdr} is the list of the remaining elements. The list of no elements is the atomic symbol \texttt{nil}.

\[
\text{car of } (a \ b \ c) = a \\
\text{cdr of } (a \ b \ c) = (b \ c) \\
\text{cdr of } (a) = \texttt{nil}
\]
The cons of an S-expression and a list is the list with the S-expression tacked onto the front.

cons of a and (b c) = (a b c)

LISP does arithmetic with the functions plus, minus, times, logor, logand, logxor, quotient, and remainder. The arguments of these functions must be numbers. Plus, times, logor, logand, and logxor take any number of arguments and calculate the sum, product, bitwise inclusive or, bitwise and, and bitwise exclusive or of all of the arguments. Minus takes one argument and returns its negative (ones complement). Quotient and remainder take two arguments and return the quotient and remainder of the integer division of the first by the second.

LISP uses the atomic symbols t and nil to represent truth and falsehood, respectively. Decisions are made by functions called predicates. A predicate applies a test to its argument and returns t or nil depending on the result.

Atom returns t if its argument is an atom (number or atomic symbol) and nil if it a concatenation.

Numberp returns t only if its argument is a number. It returns nil if it is an atomic symbol or a concatenation.

Null returns t if its argument is nil, and returns nil otherwise.

Equal takes two arguments and returns t if they have the same structure, i.e., if they would look alike if printed out.

The argument of zerop must be a number. It returns t if that number is plus zero.

Greaterp takes two numeric arguments and returns t if the first is algebraically strictly greater than the second.

Predicates are usually used with cond, the decision making function. Cond takes an indefinite number of arguments, each of which is a list of two items— an antecedent and a consequent. The arguments are examined one at a time until an antecedent is found to be true. The value of the corresponding consequent is then returned. If all of the antecedents are false an error message is printed. The antecedent of the last argument is usually t to prevent this.

Logical quantities are manipulated with the functions and and or. Each takes an indefinite number of logical arguments and returns t if all of them or one of them, respectively, is true. There is no function named not, but null may be used to negate logical quantities. Remember that logand and logor are used with numerical quantities, and and or with logical ones.
Atomic symbols have the property that they may "stand for" things. The thing for which an atomic symbol stands is called its value. It may be any S-expression, atomic or otherwise. The value of an atom may be the atom itself, as is the case with \texttt{t} and \texttt{nil}. When an atom has a value, it is said to be bound to that value. Not all atoms are bound. The predicate \texttt{valp} may be used to determine whether an atomic symbol has a value.

The function \texttt{setq} is used to bind atomic symbols. The first argument is the symbol, the second is the value. Any previous value of the symbol is lost. \texttt{setq} is an example of a pseudo-function. It is used for its effect rather than its value. Pseudo-functions, like all other functions, must return values, but the value is usually ignored. \texttt{setq} returns its second argument.

In LISP function calls are written as S-expressions, using a variation of Polish notation. The S-expression used is a list containing the function as the first item and the arguments as the remaining items. Every function must therefore be able to be written as an S-expression. For this purpose, associated with each internal function in LISP is an atomic symbol with the same name. The value of the symbol is a number with an invisible flag giving the LISP program the information it needs to call the subroutine.

Suppose, for example, that \texttt{X} stands for \texttt{a} and \texttt{Y} stands for \texttt{(b c)}, and one wishes to take the \texttt{cons} of \texttt{X} and \texttt{Y}, obtaining \texttt{(a b c)}. The atomic symbol \texttt{cons} is used, and the call is the S-expression \texttt{(cons X Y)}. Note that the \texttt{car} of a function call is the function, and the \texttt{cdr} is the argument list. The procedure by which the S-expression \texttt{(cons X Y)} is transformed into \texttt{(a b c)} is called evaluation, and is the most important procedure in LISP. Evaluation of a number gets the number itself. Evaluation of an atomic symbol gets the symbol's value. Evaluation of a nonatomic S-expression (which must be a list) causes the arguments (in most cases) to be evaluated, and their values sent to the function. Whether the arguments of a function are evaluated or not is a property of the function. Except where otherwise specified, all functions have their arguments evaluated.

Since the arguments of functions are evaluated, they may be other function calls, enabling functions to be nested within each other. For example, to take the \texttt{car} of the \texttt{cdr} of the \texttt{car} of whatever \texttt{X} is bound to, evaluate

\[(\text{car (cdr (car x))})\]

If \texttt{x} evaluates to \texttt{(1.3)}, then

\[(\text{plus (minus (car x)) 5 (cdr x)})\]

evaluates to 7.

Evaluation may be stopped with the function \texttt{quote}. \texttt{Quote} takes one argument and returns it without evaluation. To find the \texttt{cons} of \texttt{a} and \texttt{(b c)}, obtaining \texttt{(a b c)}, evaluate
(cons (quote a) (quote (b c)))

LISP will evaluate (quote a) to a and (quote (b c)) to (b c) before sending them to cons. Evaluating (cons a (b c)) would cause the value of c to be sent to the function b, and the value of a concatenated with the result.

The function setq evaluates its second argument but not its first. To bind x to (a b c), evaluate

(setq x (quote (a b c)))

x may be set to 1 more than its previous numerical value by evaluating

(setq x (plus x 1))

The function cond does not evaluate its arguments directly but evaluates the antecedents and consequents separately. The antecedent of each argument is evaluated until one is found to have a value of t. The consequent is then evaluated.

(cond ((atom x) (quote a)) (y (quote (a b)))
 (t (quote (a b c))))

evaluates to a if the value of x is atomic, or
(a b) if the value of y is t, or
(a b c) otherwise.

The third case works because t and nil are bound to themselves.

In addition to functions written as subroutines, functions may be written by the programmer. These functions are interpreted by LISP when they are called. A programmed function is a list of three items, the first of which is usually the atomic symbol lambda. Lambda is not a subroutine but a special symbol which LISP recognizes. It has no value. Like subroutines, programmed functions are usually given names, and an atomic symbol of the same name is given a value of the function. Functions defined with lambda are called expr's and always have their arguments evaluated.

Suppose the symbol foo has a value of

(lamba (x) (times 2 x x))

Lambda identifies it as an expr, (x) is the dummy symbol list, indicating that it takes one argument, and (times 2 x x) is the actual definition. This function takes one argument, squares it, multiplies by 2, and returns the result.

(foo (plus 1 2)) evaluates to 18
(foo (foo 1)) evaluates to 8
A predicate `symbp` to detect whether an S-expression is an atomic symbol could be written as

```lisp
(lambda (x) (and (atom x) (null (numberp x)))), or as
```

```lisp
(lambda (y) (cond
  ((numberp y) nil)
  ((atom y) t)
  (t nil))
)
```

This predicate, if defined, could be used exactly as one would use `atom` or `numberp`. Programmed functions may call any functions, even themselves.

A function `fact` to find the factorial of a number could be written as

```lisp
(lambda (x) (cond
  ((equal x 1) 1)
  (t (times x (fact (plus x -1))))))

(fact 4) evaluates to 24
(fact (fact 3)) evaluates to 720
```

When LISP attempts to evaluate a function call, that is, a nonatomic S-expression, it evaluates the function as many times as necessary (usually once) until a subroutine or a list beginning with `lambda`, `nlamda`, or `lambda` is found. If `lambda` is found, the arguments are evaluated and paired with the symbols on the dummy symbol list. Any old values of these symbols are saved, and the symbols are bound to the evaluated arguments. The definition is then evaluated and, after restoring the previous bindings of the dummy symbols, the value of the definition is returned as the value of the call. Example — suppose that the second definition of `symbp` is used. Find out whether `a` is a symbol.

`symbp` evaluates to `(lambda (y) (cond ((numberp y) nil)
  ((atom y) t) (t nil)))

`y` evaluates to `(1 2 3)`. This is its previous value from some unrelated calculation.
Evaluate `(symbp (quote a))`.

The function `symbp` is evaluated to

```lisp
(lambda (y) (cond ( numberp y) nil) ((atom y) t) (t nil)))
```

LISP recognizes this as an expr, so it evaluates each argument

`(quote a)` is evaluated — its value is `a`

The dummy symbol list is `(y)`. The old value of `y`, `(1 2 3)`, is saved.

`y` is bound to the evaluated argument.
`y` now evaluates to `a`.

`(cond ((numberp y) nil) ((atom y) t) (t nil))`, the definition of the function, is evaluated. Cond, unlike most functions, does not evaluate its arguments immediately, so the arguments sent to cond are

```lisp
((numberp y) nil), ((atom y) t), and (t nil).
```

`(numberp y)`, the first antecedent, is evaluated. `Numberp` evaluates its argument.
y is evaluated, obtaining a, which is sent to numberp. 
\( a \) is not a number, so numberp returns nil.
cond goes to the next antecedent.
(atom y) is evaluated.
y evaluates to a, which is an atom, so atom returns t.
The second statement is true, so its consequent, t, is 
evaluated.
t is bound to itself, so the value of the call to cond 
is t.
y is restored to (1 2 3), its old value, and t is returned 
as the value of (symbp (quote a)).
Hence \( a \) is a symbol.

Functions like symbp and fact, being values of atomic 
symbols, are relatively permanent and may be used repeatedly.
When one wishes to use a function only once, it is not 
necessary to give it a name and bind the atomic symbol of 
the same name to it. The function itself may be used. 
Hence

\[
\begin{align*}
\text{(lambda (y) (cond ((numberp y) nil) ((atom y) t) 
(t nil))) (quote a)}
\end{align*}
\]

may be used to determine that \( a \) is a symbol.

If a function is recursive, however, it must be given a 
name so that it may use that name in calling itself. An 
attempt to calculate 4 factorial by evaluating

\[
\begin{align*}
\text{(lambda (x) (cond ((equal x 1) 1) (t (times x (fact 
(plus x -1))))))) 4)}
\end{align*}
\]

would not work because fact has no value.

\[
\begin{align*}
\text{(label fact (lambda (x) (cond ((equal x 1) 1) (t (times 
x (fact (plus x -1))))))) 4)}
\end{align*}
\]

will work. Label is used to temporarily bind the atomic 
symbol fact to the definition of the factorial function.

A list consisting of label, a symbol, and a function is 
equivalent to the function alone, except that the symbol is 
temporarily bound to the function. The function can therefore 
call itself by referring to the symbol with which it is 
labelled. The previous value of the symbol is restored when 
the function is finished.

To write a function which does not have its arguments 
evaluated, use nlambda instead of lambda. Functions defined 
in this way are called fexpr's. In addition to not 
evaluating their arguments, they also have a different way 
of binding arguments to the dummy symbol list. A fexpr may 
take any number of arguments. There must be exactly one 
symbol on the dummy symbol list. It will be bound to the 
list of all of the arguments. Functions which do not 
evaluate their arguments are usually used only on the top 
level for "utility" purposes. Other functions do not, as a 
rule, call them, and they are not usually recursive. 
prindf, dex, fix, trace, and untrace are examples of
internal functions which do not evaluate their arguments.

Expr's may be conveniently defined by means of the pseudo-function dex. Dex takes three arguments and does not evaluate them. The first is the name of the function to be defined, the second is the dummy symbol list, and the third is the definition.

```
(dex symb (y) (cond ((numberp y) nil)
               ((atom y) t) (t nil)))
```

evaluates to symb and defines symb to be the expr described above.

Programs in the usual sense may be written with the functions prog, return, and go. Prog takes an indefinite number of arguments and does not immediately evaluate them. The first argument is the temporary variables list. The value of each symbol on this list is saved, and each symbol is bound to nil. These symbols may be used for temporary storage by the program, and will have their original values restored upon exit. Of the remaining arguments, atomic symbols are interpreted as address tags and nonatomic expressions as instructions. The previous values of the tags are saved, and the tags are bound to pointers to the appropriate points in the program. The instructions are then evaluated in sequence, and the values ignored. If the program runs out of instructions, nil is returned. If the function return is called, its evaluated argument is returned as the value of the program. In either case the temporary variables and address tags are restored. The function go, with an address tag as an argument, causes a transfer of control to the point indicated. Prog, like other functions, may be nested. Return and go always refer to the most recently entered prog. Program variables of nested progs are saved independently at each level. On the top level of a prog the rules for use of cond are relaxed. If cond runs out of propositions, rather than giving an error message it simply goes to the next statement of the program.

A typical use of prog is in the function reverse, which reverses a list. Reverse could be defined by

```
(dex reverse (x) (prog (y)
   z (cond ((null x) (return y)))
   (setq y (cons (car x) y))
   (setq x (cdr x))
   (go z)
))
```

This takes advantage of the facts that the program variable is initially bound to nil and that a cond may run out of propositions in a prog.

LISP keeps a symbol table similar to those used by debuggers and assemblers, containing an entry for each symbol, whether it has a value or not. This table initially contains
one entry for each subroutine, with a name the same as that of the subroutine and a value of a number with the invisible flag subr or fsubr.

\texttt{t} and \texttt{nil}, with values of themselves

\texttt{lambda}, \texttt{mlambda}, \texttt{label}, \texttt{subr}, and \texttt{fsubr}, with no values. These are flags used internally by the LISP program.

Whenever an atomic symbol not appearing in the table is read in, it is placed in the table with no value. It may later be given a value.

The value of an atomic symbol is stored on the \texttt{car} of that symbol. Taking the \texttt{car} of an atomic symbol gets its value. It is illegal to take the \texttt{car} of a symbol with no value, and it is very dangerous to take the \texttt{car} of a number. The \texttt{cdr} of a symbol is normally \texttt{nil}, but any S-expression may be stored there by the programmer.

S-expressions which look alike may occupy different locations of memory. Expressions may also be different but share common sub-expressions. Whenever an S-expression is read in, a fresh copy of it is created in memory, even if another copy already exists. Only atomic symbols are in unique locations. For example, reading and evaluating

\begin{verbatim}
(setq x (quote ((a.b) (c.d)))) and
(setq y (quote ((a.b) (c.d))))
\end{verbatim}

leaves memory looking like this -

![Diagram of S-expression structure]

\textbf{fig. 3 - non-identical S-expressions}

\texttt{x} and \texttt{y} will both print out as \texttt{((a.b) (c.d))}, and will satisfy the predicate \texttt{equal}, but they will not be identical. Another predicate, \texttt{eq}, may be used to test for exact identity between two S-expressions. In the example above, \texttt{(eq x y)} would evaluate to \texttt{nil}. If \texttt{x} and \texttt{y} had been bound by

\begin{verbatim}
(setq x (quote ((a.b) (c.d)))) and
(setq y x)
\end{verbatim}

they would be identical and would satisfy \texttt{eq}. 
Equal could have been written in terms of eq as

\[
(dx \text{ equal} \ (x\ y) \ (cond
\quad ((\text{and} \ (\text{numberp} \ x) \ (\text{numberp} \ y)) \ (\text{zerop} \ (\text{logxor} \ x\ y)))
\quad ((\text{or} \ (\text{numberp} \ x) \ (\text{numberp} \ y)) \ \text{nil})
\quad ((\text{atom} \ x) \ (\text{atom} \ y)) \ (\text{eq} \ x\ y)
\quad ((\text{or} \ (\text{atom} \ x) \ (\text{atom} \ y)) \ \text{nil})
\quad (t \ (\text{and} \ (\text{equal} \ (\text{car} \ x) \ (\text{car} \ y)) \ (\text{equal} \ (\text{cdr} \ x) \ (\text{cdr} \ y)))
\))
\]

There are two S-expression modifying functions, replace and replacd, each taking two arguments and evaluating both. They replace the car and cdr, respectively, of the first argument with the second argument, and return the modified first argument. If x and y are bound as in figure 3, (replace (car x) (quote q)) removes the dotted line in the figure and replaces it with a pointer to the atomic symbol q. It returns (a.q), its modified first argument. The value of x is now ((a.q) (c.d)). The value of y is not changed. If y had been bound by (setq y x), so that its value was identical with that of x, it too would have been changed. Since the car of an atomic symbol is its value, replace may be used to bind symbols, and replacd may be used to store S-expressions on the cdr of a symbol.

Other S-expression manipulating functions

Caar, cadr, cdar, and cddr are compositions of car and cdr. They could have been defined by

\[
(dx \ caar \ x) \ (\text{car} \ (\text{car} \ x))\\
(dx \ cadr \ x) \ (\text{car} \ (\text{cdr} \ x))\\
(dx \ cdar \ x) \ (\text{cdr} \ (\text{car} \ x))\\
(dx \ cddr \ x) \ (\text{cdr} \ (\text{cdr} \ x))
\]

For example, the cadr of (a b c) is the car of the cdr of (a b c), or b.

List takes (and evaluates) an indefinite number of arguments and returns the list of them. List and cons are the two functions that are used to create complex S-expressions out of small ones.

\[
(\text{list} \ 1 \ (\text{cons} \ \text{quote} \ a) \ (\text{quote} \ b)) \ (\text{plus} \ 1 \ 2 \ 3)
\]

Evaluates to (1 (a.b) 6)

Append takes two arguments, which must be lists, and appends them.

\[
(\text{append} \ (\text{quote} \ (a\ b\ c)) \ (\text{quote} \ (d\ e\ f)))
\]

Evaluates to (a b c d e f)

Append makes a copy of the first list in order to avoid modifying it. Append could have been defined by

\[
(dx \ \text{append} \ (x\ y) \ (\text{cond}
\quad ((\text{null} \ x) \ y)
\quad (t \ (\text{cons} \ \text{car} \ x) \ (\text{append} \ (\text{cdr} \ x) \ y)))
\]
Nconc is the same as append except that it does not copy its first argument but merely changes the nil at the end of the first list to the second list. In doing so the first list is permanently modified. Nconc could have been defined by

\[
\begin{align*}
\text{(dex nconc (x y) (cond} \\
\text{ (null x) y} \\
\text{ (t (prog (z) \\
\text{ (setq z x) \\
\text{ (cond ((null (null (cdr z))) (go b)) \\
\text{ (rplacd z y) \\
\text{ (return x)) \\
\text{ (go a)) \\
\text{))))))))}}
\end{align*}
\]

Reverse takes one argument, which is a list, and reverses it. It could have been defined by

\[
\begin{align*}
\text{(dex reverse (x) (prog (y) \\
\text{ a (cond ((null x) (return y))) \\
\text{ (setq y (cons (car x) y)) \\
\text{ (setq x (cdr x)) \\
\text{ (go a)) \\
\text{))})}}
\end{align*}
\]

Subst takes three arguments and substitutes the first for all appearances, on all levels, of the second in the third. The third argument is not actually modified. Subst could have been defined by

\[
\begin{align*}
\text{(dex subst (x y z) (cond} \\
\text{ ((equal y z) x) \\
\text{ ((atom z) z) \\
\text{ (t (cons (subst x y (car z)) (subst x y (cdr z)))) \\
\text{))))}}
\end{align*}
\]

Sassoc takes three arguments and looks up the first in the second, which is a special type of table called an association list. An association list is a list of dotted pairs of atomic symbols with the S-expressions associated with them. For example, to keep a table with the information

\[
x=1 \ y=2 \ z=3
\]

and not bind \(x\), \(y\), and \(z\) to these values, one could bind \text{tab} to

\[
((x.1) \ (y.2) \ (z.3))
\]

Sassoc can look through a table in this format. It returns the first pair which has a \text{car} identically equal to the first argument.

\[
(\text{sassoc (quote y) tab no}) \text{ evaluates to (y.2)}
\]
The third argument is a function of no variables which is called if the item is not found.

\[(\text{assoc} \ (\text{quote} \ z) \ \text{tab} \ (\text{quote} \ (\lambda \ x \ \text{nil} \ (\text{quote} \ \text{not found}))))\]

evaluates to \((z.3)\). If \(z\) had not been found, \((\text{not found})\)
would have been returned as the value of the call to \text{assoc}. In order to save space in memory, a number may be used as
the third argument. If the search fails the \text{uaf} error
message will be printed along with the number. \text{assoc}
could have been written as

\[
(\text{dex assoc} \ (x \ y \ z) \ (\text{cond}
\begin{align*}
&((\text{null} \ y) \ (z)) \\
&\{\text{eq} \ x \ (\text{caar} \ y)\} \ (\text{car} \ y) \\
&\text{t} \ (\text{assoc} \ x \ (\text{cdr} \ y) \ z)\}
\end{align*}
))
\]

Other predicates

\text{Member} takes two arguments, the second of which is a
list, and returns \text{t} if the first argument is a member of
that list. \text{Equal} is used for the comparison, so any S-
expression may be tested. The second argument is searched
on the top level only. \text{Member} could have been defined by

\[
(\text{dex member} \ (x \ y) \ (\text{cond}
\begin{align*}
&((\text{null} \ y) \ \text{nil}) \\
&\{\text{eq} \ x \ (\text{car} \ y)\} \ \text{t} \\
&\text{t} \ (\text{member} \ x \ (\text{cdr} \ y))\}
\end{align*}
))
\]

I/O operations

\text{Read} takes no arguments. It reads one S-expression from
the typewriter or tape reader and returns that S-expression.

\((\text{read})\) evaluates to \((a \ b \ c \ d)\) if the latter is typed in.

\text{Print} takes one argument, which must be an atom. It
prints and/or punches it with no extra punctuation. The
value returned is the original argument.

\text{Print} takes one argument, which may be any S-expression.
It prints and/or punches it preceded by a carriage return
and followed by a space. The argument is returned.

\[(\text{print} \ (\text{quote} \ (a \ b \ c)))\] prints out "\(a \ b \ c\)"
and returns \((a \ b \ c)\)

\text{Terpri} prints and/or punches a carriage return. It takes
no arguments and returns \text{nil}.

\text{Stop} takes no arguments. It waits for a character to be
typed on the typewriter and then returns \text{nil}. A call to
stop is normally punched at the end of each tape in order to
give the operator time to load a new tape or change sense
switches.

Miscellaneous functions

gensym takes no arguments. Each call to gensym creates a
new atomic symbol as if it had been read in and returns that
symbol. The names of the symbols are g00001, g00002, etc.

eval takes one argument and returns its value. This
means that the argument is actually evaluated twice. If x
is bound to (cons 1 3), the value of (eval x) is (1 3),
whereas the value of x alone is (cons 1 3).

apply takes two arguments, a function and an argument
list for the function. The function is called with the
given arguments. If the function is one which normally
evaluates all its arguments, they will not be evaluated
again, but simply taken from the second argument to apply,
which was, of course, evaluated already.

(apply (quote cons) (quote (a b))) sends a and b,
without further evaluation, to cons, thereby returning
(a b).

trace takes any number of arguments and does not evaluate
them. Each argument should be the name of an expr (function
using lambda). Each function is traced, or modified so that
it prints out its name and evaluated arguments on entry, and
its name and returned value on exit. Nested or recursive
functions cause the printouts to occur in proper order at
each entry and exit.

If fact initially had a value of

(lambda (x) (cond ((equal x 1) 1) (t (times
  x (fact (plus x -1))))))

(trace fact) would evaluate to t and redefine fact as

(lambda (x) (prog (99g)
  (print (quote (enter fact)))
  (print (list x))
  (setq 99g
    (cond ((equal x 1) 1) (t (times
      x (fact (plus x -1))))))
  (print (quote (value fact)))
  (return (print 99g))))

Evaluation of (fact 3) would return 6 after printing

(enter fact)
(3)
(enter fact)
(2)
(enter fact)
(1)
(value fact)
1
(value fact)
2
(value fact)
6

**Untrace** takes any number of arguments and does not evaluate them. Each argument should be the name of a traced function. **Untrace** restores each function to its original definition.

**Prindef** is used to punch out the definitions of functions and constants. It takes any number of arguments and does not evaluate them. Each argument should be an atomic symbol with a value. **Prindef** punches the definition of each symbol as a call to **rplaca**, and then returns a call to **stop**, which is normally punched also. The resultant tape, when read in at a later time, defines the atoms and then waits for a character from the typewriter.

**(prindef fact)** punches

**(rplaca (quote fact) (quote (lambda (x) (cond ((equal x 1) 1)
(t (times x (fact (plus x -1)))))))

**(stop)**

**Prindef** could have been defined by

**(setq prindef (quote (nlambda (x) (prog nil
a (cond ((null x) (return (quote (stop)))
(print (list
 (quote rplaca)
 (list (quote quote) (car x))
 (list (quote quote) (eval (car x)))
))
 (terpri)
 (setq x (cdr x))
 (go a)
))))

**Fix** is used to edit or modify functions. It takes three arguments and does not evaluate them. The third is the name of the function to be fixed. The first argument is substituted for the second in each appearance in the function, and the function is redefined to be the result. **Fix** could have been defined by

**(setq fix (quote (nlambda (x)
 (rplaca (car (cdr x)) (subst (car x) (car (cadr x))
 (eval (car (cadr x))))))
)))

**Prog2** is used to cause the evaluation of two functions with a single call to **eval**. It takes two arguments, evaluates both, and returns the second. **Prog2** could have
been defined by
\[(\text{dex prog2} \ (x \ y) \ y)\]

\textsf{Nconc} could have been written more efficiently using \textsf{prog2} -
\[(\text{dex nconc} \ (x \ y) \ (\text{cond}
\quad ((\text{null} \ x) \ y)
\quad (\text{t} \ (\text{prog} \ z)
\quad (\text{setq} \ z \ x)
\quad (\text{a}
\quad (\text{cond} \ ((\text{null} \ (\text{cdr} \ z)) \ (\text{prog2} \ (\text{rplacd} \ z \ y) \ (\text{return} \ x))))
\quad (\text{setq} \ z \ (\text{cdr} \ z))
\quad (\text{go} \ a)
\quad )))\)]

\textsf{Maplist} is used to send each item of a list to a function as the single argument of that function, and return the list of the values returned. \textsf{Maplist} takes two arguments and evaluates both. The first is the list of arguments, the second is the function. To \textsf{cons} each item of the list \((a \ b \ c \ d)\) with \(x\), for example, evaluate
\[(\text{maplist} \ (\text{quote} \ (a \ b \ c \ d)) \ (\text{quote} \ (\text{lambda} \ (y) \ (\text{cons} \ y \ (\text{quote} \ x))))))\]

obtaining
\[((a.x) \ (b.x) \ (c.x) \ (d.x))\]

\textsf{Maplist} could have been defined by
\[(\text{dex maplist} \ (x \ y) \ (\text{cond}
\quad ((\text{null} \ x) \ \text{nil})
\quad (\text{t} \ (\text{cons} \ (\text{apply} \ y \ (\text{list} \ (\text{car} \ x))) \ (\text{maplist} \ (\text{cdr} \ x) \ y)))\]

Output

Output normally goes to the online typewriter. If sense switch \(3\) is up output goes to the punch also. Sense switch \(6\) independently suppresses type-out. The punch is automatically assigned and dismissed as needed. Error messages are always printed on the typewriter only.

\textit{S}-expressions which are nearly lists, such as
\[\text{(a.(b.(c.d)))}\]

are printed as
\[\text{(a b c.d)}\]

This format is also acceptable for input.

Numbers are printed as signed integers, in octal if sense switch \(4\) is up, in decimal otherwise. Sense switch \(4\) is
interrogated only after reading or printing a number.

A carriage return is printed after every 63 characters of output not containing a carriage return.

Input

Input comes from the tape reader if sense switch 5 is down and from the typewriter if up. The reader is automatically assigned and dismissed as needed. A call to subroutine stop always clears the time-sharing reader buffer. After turning off sense switch 5 it is necessary to type a carriage return to start reading tape.

Carriage return and stop code are ignored.

Parentheses, period, and space separate atoms. Extra spaces may be used anywhere except inside an atom. Spaces may be omitted except where needed to separate atoms. Tab and comma are equivalent to space. () is equivalent to nil. When an S-expression consists of an atom only it must be followed by a separation character, usually space. This separator is saved and used on the next call to read.

0 to 7 in octal radix (sense switch 4 up) and 0 to 9 in decimal radix are digits. An atom containing only digits, with an optional minus sign, is a number. Plus signs are not permitted in numbers. The absolute value of a number must not exceed 131071 decimal or 377777 octal.

Other characters, including case shifts and all uppercase characters, are letters, and atoms containing one or more letters are atomic symbols. All atoms must begin and end in lower case. Atomic symbols are limited to six characters plus a lower case shift at the end if needed. Atomic symbols longer than this are truncated.

Backspace may be used to correct errors in typing. After one or more characters of an atom have been typed, backspace deletes those characters and starts the atom over. The remainder of the S-expression is not affected. A backspace immediately after a separation character starts the entire S-expression over and prints out a carriage return.

Operation

Read in the tape, set the sense switches as desired, and start at zero. LISP reads S-expressions and prints out their values. The LISP program could be simulated by

(progn nil a (print (eval (read))) (go a))

Some other LISP programs, notably the version used on the 7094, use a different algorithm, in which a function and its argument list are typed in as two separate S-expressions, and the arguments are not evaluated on the top level. This algorithm may be approximately simulated by

(progn nil a (print (apply (read) (read))) (go a))
When first starting LISP, if sense switch 2 is on, core 1 is assigned and used. About three times as much free storage is available when using 8K as when using 4K.

If sense switch 1 is on when LISP is started, functions may be deleted, resulting in more available free storage and symbol table space. Subroutines may be deleted only in a specified order, and deletion of any subroutine requires deletion of all that precede it. After LISP prints out each subroutine name, it listens for a character from the typewriter. If "x" is typed, the subroutine is deleted and LISP prints the next one. If any other character is typed, the subroutine is not deleted, and LISP begins normal operation. The order in which subroutines may be deleted is

trace (deletes untrace also), reverse, fix, subst, dex, prindfr, sassoc, gensym, member, noonc, append, maplist, or, and, quotient, remainder, greaterp, logxor, logor, logand, times, plus, minus, equal, and eq.

LISP may be stopped at any time except during a garbage collection by hitting call and starting at location zero. Temporary bindings that are in effect at that time will not be removed, but this rarely causes difficulty. Starting at zero during a garbage collection will usually destroy most of free storage. LISP indicates that it is garbage collecting by turning on the coordinate lights on the cathode ray display.

LISP may execute an illegal instruction if an improper operation is performed, such as an attempt to bind a number. Starting at zero is usually safe in this case.

Upon detection of an error, LISP prints a 3-letter error code on the typewriter, sometimes followed by the S-expression in error. Except in the case of the pce and sce errors, the computation continues.

uas (unbound atomic symbol) - The argument of a call to eval is an atomic symbol with no value. The symbol in error is printed. nil will be returned as the value of the call.

uaf (unbound atomic function) - A number without the subr or fasubr flag, or a symbol which is not bound or is bound to itself, has been used as a function. The number or symbol is printed. The arguments for the function will not be evaluated, and nil will be returned.

tfa (too few arguments) - A subr or expr has not been given enough arguments, or the symbol list after nlambda contains more than one symbol. nil will be returned.

tma (too many arguments) - A subr or expr has been given
too many arguments, or the symbol list after \texttt{nilama} is empty. \texttt{Nil} will be returned.

cva (car of valueless atom) - an attempt has been made to calculate the \texttt{car} of an atomic symbol which has no value. The symbol in error is printed, and \texttt{nil} will be returned.

\texttt{icd} (illegal conditional) - A call to \texttt{cond} has run out of propositions. \texttt{Nil} will be returned.

\texttt{ana} (argument not atomic) - The argument to \texttt{print} or \texttt{valp} is not atomic. \texttt{Nil} will be returned.

\texttt{nna} (non-numeric argument) - An argument to \texttt{plus}, \texttt{times}, \texttt{logor}, \texttt{logand}, \texttt{logxor}, \texttt{quotient}, \texttt{remainder}, \texttt{zerop}, or \texttt{greaterp} is not a number. It will be taken as zero.

\texttt{ovf} (overflow) - The second argument for \texttt{quotient} or \texttt{remainder} is zero. Zero will be returned.

\texttt{ilp} (illegal parity) - A character from the tape reader has even parity. It will be ignored.

\texttt{bsy} (busy) - The reader, punch, or core 1 is busy. Type any character to try again.

\texttt{pce} (pushdown capacity exceeded) - The combined length of the pushdown list and symbol table is too great. LISP starts over at location zero. All temporary bindings remain.

\texttt{sce} (storage capacity exceeded) - The free-storage list has been exhausted, and no space could be reclaimed by the garbage collector. LISP starts over as with \texttt{pce}.

\texttt{iif} (illegal input format) - An object which is not an S-expression has been read. The entire call to \texttt{read} will be started over.
| name    | type  | number of args | description | type          | name    | type  | number of args | description | type          |
|---------|-------|----------------|-------------|---------------|---------|-------|----------------|-------------|---------------|---------------|
| car     | subr  | 1 e            | general     |               | cdr     | subr  | 1 e            | general     |               |
| caar    | subr  | 1 e            | car·car     |               | caadr   | subr  | 1 e            | car·cdr     |               |
| cdar    | subr  | 1 e            | general     |               | cdadr   | subr  | 1 e            | general     |               |
| cons    | subr  | 2 e            | general     |               | list    | fsubr | n e            | general     |               |
| rplaca  | subr  | 2 e            | PF general  | y → (car x)   | rplacd  | subr  | 2 e            | PF general  | y → (cdr x)   |
| append  | subr  | 2 e            | general     |               | nconc   | subr  | 2 e            | PF general  | (append x y) → x|
| reverse | subr  | 1 e            | general     |               | subst   | subr  | 3 e            | general     | subst x for y in z |
| sassoc  | subr  | 3 e            | general     | look up x in y, or call z |
| and     | fsubr | n e            | logical     | x and y and z ... |
| or      | fsubr | n e            | logical     | x or y or z ... |
| null    | subr  | 1 e            | predicate   | x = nil       |
| atom    | subr  | 1 e            | predicate   | x is atom     |
| numberp | subr  | 1 e            | predicate   | x is number   |
| valp    | subr  | 1 e            | predicate   | x is bound    |
| zerop   | subr  | 1 e            | predicate   | x = 0         |
| greaterp| subr  | 2 e            | predicate   | x > y         |
| eq      | subr  | 2 e            | predicate   | x is y exactly |
| equal   | subr  | 2 e            | predicate   | x looks like y|
| member  | subr  | 2 e            | predicate   | x is a member of y|

**Description:**
- **car:** Returns the first element of a list.
- **cdr:** Returns the rest of a list.
- **caar:** Evaluates the first element of a list.
- **cadr:** Evaluates the rest of a list.
- **cdar:** Evaluates the second element of a list.
- **cons:** Constructs a list.
- **list:** Constructs a list.
- **rplaca:** Replaces the first element of a list.
- **rplacd:** Replaces the rest of a list.
- **append:** Appends two lists.
- **nconc:** Negates the append operation.
- **reverse:** Reverses a list.
- **subst:** Substitutes a value in a list.
- **sassoc:** Searches for a value in a list.
- **and:** Returns true if all arguments are true.
- **or:** Returns true if any argument is true.
- **null:** Checks if a list is empty.
- **atom:** Checks if a list is a symbol.
- **numberp:** Checks if a list is a number.
- **valp:** Checks if a list is a value.
- **zerop:** Checks if a list is zero.
- **greaterp:** Checks if a list is greater than another.
- **eq:** Checks if two lists are equal.
- **equal:** Checks if two lists look the same.
- **member:** Checks if a list is a member of another list.
<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>fsubr</td>
<td>arith</td>
</tr>
<tr>
<td>minus</td>
<td>subr</td>
<td>arith</td>
</tr>
<tr>
<td>times</td>
<td>fsubr</td>
<td>arith</td>
</tr>
<tr>
<td>logor</td>
<td>fsubr</td>
<td>arith</td>
</tr>
<tr>
<td>logand</td>
<td>fsubr</td>
<td>arith</td>
</tr>
<tr>
<td>logxor</td>
<td>fsubr</td>
<td>arith</td>
</tr>
<tr>
<td>quotient</td>
<td>subr</td>
<td>arith</td>
</tr>
<tr>
<td>remainder</td>
<td>subr</td>
<td>arith</td>
</tr>
<tr>
<td>read</td>
<td>subr</td>
<td>PF I/O</td>
</tr>
<tr>
<td>print</td>
<td>subr</td>
<td>PF I/O</td>
</tr>
<tr>
<td>print</td>
<td>subr</td>
<td>PF I/O</td>
</tr>
<tr>
<td>terpri</td>
<td>subr</td>
<td>PF I/O</td>
</tr>
<tr>
<td>stop</td>
<td>subr</td>
<td>PF I/O</td>
</tr>
<tr>
<td>gensym</td>
<td>subr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>quote</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>setq</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>cond</td>
<td>fsubr</td>
<td>misc.</td>
</tr>
<tr>
<td>eval</td>
<td>subr</td>
<td>misc.</td>
</tr>
<tr>
<td>apply</td>
<td>subr</td>
<td>misc.</td>
</tr>
<tr>
<td>trace</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>untrace</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>prindef</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>dex</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>fix</td>
<td>fsubr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>prog</td>
<td>fsubr</td>
<td>misc.</td>
</tr>
<tr>
<td>go</td>
<td>subr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>return</td>
<td>subr</td>
<td>PF misc.</td>
</tr>
<tr>
<td>maplist</td>
<td>subr</td>
<td>misc.</td>
</tr>
<tr>
<td>pro2</td>
<td>subr</td>
<td>misc.</td>
</tr>
</tbody>
</table>

- **arith**: Arithmetic operations
- **PF**: Procedures
- **misc.**: Miscellaneous operations

- **x + y + z ...**
- **-x**
- **x \times y \times z ...**
- **x \lor y \lor z ...**
- **x \land y \land z ...**
- **x \sim y \sim z ...**
- **[x/y]**
- **x - y \times [x/y]**
- **print atom**
- **print S-expression**
- **print carriage return**
- **wait**
- **create symbol**
- **bind x to y**
- **value of x**
- **call x with y**
- **print definitions**
- **define expr**
- **fix x for y in z**
- **go to x**
- **return from program**
- **send each element of x to y**
- **y**
lisp for ts 1/1/66

test=sas hih

define error who, where
q=f lexo who
jsp err'where
[ qA77x1 ] M [ qA7700 ] M [ qA770000x100 ]
terminate

ci=(lt
ct=(1
ctn=(nil
cl=(1

c3=(3

cpde=(pde

repeat 1-if2, equals halt, stop
bind=jdp bn
push=jda pw1
pop=jsp po
zorch=jdp zo

0/
jmp begin
lak rd1
dap rdx

loop,
dzm pa3
dzm pa4
lak pde
dap pd1
cal read
cal eval
cal print
jmp loop

lac 100

/ push

pw1,
dap psx
lak 1 1
adm pd1
sad snd
jmp pce
lac pw1
dac 1 pd1

jmp .

/ pop

psx,
dap pox

/lac pde
dac pw1
idx pd1
lac pw1

pox,
jmp .
bn, t4, 0 /bind
push \begin{code}
lac i pwl
dio i pwl /value to bind it to
push /old value
jmp 1 bn
\end{code}
caddr, add (1 \texttt{"caddr"})
cdar, cal car \texttt{"cdar"}
cdr, idx 100 \texttt{"cdr"}
quote, \texttt{"quote"}
car, lac i 100 \texttt{"car"}
sza i
jmp cva
x, dac 100 \texttt{main return routine}
pop ral is
spa
jmp pwl-1 / jmp
lio pwl /S-expression
pop dio i pwl
jmp x 1
cadr, add (1 \texttt{"cadr"})
caar, cal car \texttt{"caar"}
jmp car
number, sma \texttt{numberp}
expr \texttt{false sign bit}
jmp fal
sub cpe
atom, sma \texttt{atom atoms have sign bit on}
jmp fal
tru, lac ct -- point T
jmp x
zerop, cal vag \texttt{zero value}
sza
jmp fal
jmp tru
g4, t1, 0
g1, t2, 0
repeat ifm 100-., [printx /No room. ]
101/ dapr poux
sub (1
dap . 1
lac .
dap . 4
lac poux /push return jmp
push
lac 100
jmp .
cvar, error cva, -2
jmp u2
vag,
  sma
  jmp nna - s expression
  sub cpde
  sma
  jmp nna - in control table
lac i 100 - effectively can of ang = value
  x0 , t1  
  jmp x

zo, t3,
  0  /zorch add to random list
  idx i pd1
  dac t2
  (a + d)
  idx t1
  lac i t2
  dac i t1
  dio i t2
  dio i pd1
  idx pwl
  lac i pd1
  wth t with jack
  jmp 1 zo

valp,
  sma
  jmp ana
  cla
  sas i 100
  jmp tru
  fal,
  lac cn   rel
  jmp x
ana,
  error ana,-2
  jmp u2
nna,
  error nna,-2
  cal print
nnx,
  claVclivstf 4
  jmp x
in,     stf 4
       sze 50
       jmp tin
ras,   skp 600  /skip if reader not assigned
       jmp ra2
       law 51
       jdp asg
       dap ras
ra2,   rpa
       rir 7s
       sni i
       spi
       jmp in
       ril 7s
       lai
       ior (rar
       dac .2
       law 2525
       0
       spa
       jmp gtc
       error ilp
       jmp in

tin,   law i 51 /entry for stop
       xct ras
       arq    /dismiss reader
       law 600
       dap ras
       tyi
       gtc,
       lai
       and (77
       sas (74   /upper case
       sad (72   /lower case
       dac cas
       jmp x

asg,   0
       arq
       jmp bsy
       cla
       jmp 1 asg
bsy,   error bsy
       stf 4
       tyi
       law i 2
       adm asg
       jmp 1 asg
stop=.  cal tin    /"stop"
       jmp fal

cas,   72
p3, lac 1 a3
p2, cal out
out, law 77
and 100
sas cas
sad (76)
jmp oux
sad (77)
ou4, dac pcc
ior (ral)
dac oug
law 252
oug, 0
and (200
adm oug
lia
sas 1 34
jmp ou1
pas, skp 600
jmp ou2
law 47
jdp asg
dap pas
ou2, lio oug
ppa
jmp ou3
ou1, law 1 47
xct pas
arq
law 600
dap pas
ou3, szs 1 64
tyo
law 77
and 100
sas (74
sad (72
jmp oux-1
sas (56
sad (40
jmp oux
law 1 1
adm pcc
sma
jmp oux
law 77
jmp ou4
dac cas
oux, lac 100
rar 6s
jmp x
pcc, 0

/skip if punch not assigned
read,  clavstf 5  /* read */
push
lac pd1
dap re2
jmp rdx

iif,  error iif
re2,  law .  /old pd1
dap pd1
cal terpri
stf 5

rd1,  clf 6  /on if letter seen
cif 3  /on if minus sign seen
dzm a1  /value of number
lac snd
dac sy2
dac sy1
dap pt1
idx sy2
sub pd1
add (3
sma
jmp pce
lac (add-7  /character count
dac t2
llo (767676
dio i sy1
dio i sy2

rlp,  cal in
  llo cas
rir 2s
law tb1  /lower case origin
spi 1
law tb2  /upper case origin
dap tbs

tb0,  law 77
and 1 tbs
sad 100
jmp tbs
idx tbs
sas (lac tb3
jmp tb0
lac 100
sub rad
sma
jmp rsl

num,  lac a1
mul rad
scr 1s
lai
llo 100
rir 5s
spi 1
add 100
dac a1
jmp rsl 1

min,  stf 3  /-
jmp rsl 1

bsp,  lac i sy1  /backspace
sad (767676
jmp re2
jmp rd1
tb1, 20+100xnum
      54+100xmin
      55+100xpr+add
      57+100xpr+add
      73+100xper+add
      00+100xrd1+add
      33+100xrd1+add
      36+100xrd1+add
tb2, 56+100xvb
      75+100xbsp
      13+100xrlp
      77+100xrlp
tb3,vb, cal in
rs1, str 6 /letter seen
      isp t2 /pack character
      jmp rlp
      sad (add-3
      idx pt1
pt1, lac .
      lio 100
      rcr 6s
      dac 1 pt1
      jmp rlp
tbs, lac .
      lia
      rcr 6s
      dap rdx
      spi 1
      jmp rdx
      law 1 4000
      adm rdx
      lac 1 sy1
      sad (767676
rdx, jmp . /no atom
      szf 5 i
      jmp 11f
      cal mka
      jmp rxy+2
putob, law sym /oblist lookup
      dap pt1
sy1, lac .
      sas i pt1
      jmp id1-1
      idx pt1
sy2, lac .
      sas i pt1
      jmp id1
fou, idx pt1
      add (lac
      jmp x
      idx pt1
id1, law 3
      adm pt1
      sas snd
      jmp sy1
      idx pt1
dac snd
      idx snd
      idx snd
      dzm i snd
idx snd
lac cn
dac i snd
idx snd
jmp fou
1pr, szf 1i5 /*
jmp ii1f
lac cn
push
jmp rd1
per, lac 1 pd1 /*
sad cn
jmp ii1f
rar 1s
spq 5
jmp ii1f
idx 1 pd1
jmp rd1
rpr, law rd1 /*
dap rdx
lac 1 pd1
rar 1s
spq
jmp ii1f
pop
szf 5
sad cn
jmp rxy
idx pw1
lio cn
lac 1 pw1
dio 1 pw1
rxy, stf 5
dac 100
pop
sza 1
jmp x 1
push
rar 1s
spa
jmp rd5
lac 100
cal cons-1
lac 1 pd1
sad cn
jmp rdn
zorcn
jmp rdx
rdn, idx t1
dio 1 t1
jmp rd7
rd5, lio 1 pw1
lac 100
dac 1 pw1
c1f 5
rd7, dio 1 pd1
jmp rdx
mka, sas (547676) /make atom
szf 6
jmp putob /atomic symbol
cal p10 /number
szf 3
cma
crn,        lio cn   /create number
dio g1
cal cons 1
add (add
jmp x

pce,        law pde
dap pdl
error pce
jmp 0

fre,        0

snd,        lac esy

err-2,      lio 100
dio a1
err,
dap erx
c1f 4
cal terpri
lac 1 erx
cal p3
cal terpri
idx erx
lac a1

erx,        jmp .
terpri,  
cal print  
    law 7772 1 /*terpri*/
cal p2  
jmp fal  

print,  
dac t1 1 /*print*/
cal terpri  
cla  
push  
lac t1  

pn1,  
spa  
jmp pn2 1 goto return to print  

pn5,  
cal p2 1 15 expression  
lac t1  
cal cdr  
push  
lac 1 t1 1 set the car  
dac t1  
jmp pn1  

pn2,  
cal print 2  
pn6,  
pop  
dac t1  
sza 1  
jmp pn7 1 done  

law 72  

lio t1  
spi 1  
jmp pn5  
lac t1  
sad cn  
jmp pn3  
law 7372 1  
cal p2  
lac t1  
cal print 2  

pn3,  
law 5572 1  
cal p2  
jmp pn6  

pn7,  
law 72  
cal p2  

lac ai jmp p10-1( 
jmp -
```
cons-2,  cal eval-1
cons-1,  lio cn
cons,    dzn g1
         lac fre
         sza 1
         jmp gc
con2,    dac t1 /garbage collect
         lac 100
         dac 1 fre
         idx fre
         lac 1 fre
         dio 1 fre
         dac fre
         lac t1
         lia
         jmp x

null=,   xor cn          /*null*/
         jmp zerop 1
setq,    push
         cal eval-2 (eval b)
         lia     value [1]
         pop
         cal car       car of the atom is the value
         dio 1 100
         jmp prog2
rplacd,  idx 100          /*rplacd*/ (replaced a, b)
         sub (1
rplaca,  dio 1 100        /*rplaca*/
         jmp x

evlis-1, lac a2
evlis,   list,
         szf 2          /*list*/  flg x 0 -> rop
         sad cn
         jmp x
         push
         cal cons-2
         lac 1 pd1
         dac pwl
         dio 1 pd1
         jmp el2
         push
         cal cons-2
         pop

ele,     push
         cal cons-2
         pop
         zorch
         jmp ele
         sas cn
         jmp ele
         lio cn

e12,     pop
         szf 2
         idx pwl
         lac 1 pwl
         dio 1 pwl
         jmp x

e15,     pop
         szf 2
```
gfr,
dap gfx /list marker
lac 1 pt1
ral 1s
spq
jmp gfx
law 1 1
and 1 pt1
c1vswp
in1,
dac g1
in2,
dac g3
idx g3
in3,
d1o g2
d1o g4
idx g4
lac i g4
and (dip
sza i
jmp gcn
lac g1
sza i
gfx,
jmp .
lac i g3
ral 1s
spa
jmp gcb
lac i g3
and (¬dip
lia
lac i g1
lor (dip
dac i g3
lac g2
dac i g1
jmp in3
gcb,
l1o g1
lac i g3
and (¬dip
dac g1
lac g2
lor (lac
dac i g3
lac g1
jmp in2
gcn,
lac g2
sma
jmp gcl
sub cpde
sma
jmp gfx-2
l1o i g4
lac g1
lor (dip
dac i g4
in4,
lac g2
jmp in1
gcl,
l1o i g2
lac (xct
adm i g4
lac g1
dac i g2
jmp in4
gc, 
clcv lia

dpy 400
law 100
dap pt1
lac gl
sza 1
jsp gr
law sym
dap pt1

oblp,

law 2
adm pt1
jsp gr
idx pt1
jsp gr
idx pt1
sas sng
jmp oblp
lac pd1
dap pt1

pdlp,

jsp gr
idx pt1
sas el1
jmp pdlp

lac a2

low,

law frs
dac t1

swlp,

idx t1
lac 1 t1
lia
and (-lac
dac 1 t1
ril 1s
spi
jmp swlf
lac fre
dac 1 t1
law 1 1
add t1
dac fre

swlf,

idx t1
test
jmp swlp
clavcl1
dpy 300
lio a1
lac fre
sza
jmp con2
error sce
jmp 0
prog2,  lai x   /*prog2*/
    jmp x

return, dac pa3   /*return*/
go,   dac pa4   /*go*/
jmp x

prog,   lac 1 a1   /*prog*/
sad cn
    jmp pr2
dac 100   /*get a prog variable*/
lac i 100
lio cn
bind
lac 100
cal cdr
    jmp prog 1

cal cdr

pr2,   lac a1
pr3,
sad cn
    jmp pr35
lia
cal car
spa
bind
lai
    jmp pr3

pr35,   lac pa3
    push
lac pa4
    push
dzm pa3
lac a1
pr4,   cal cdr
dac pa4
sad cn
    jmp pr6   /*program finished*/
lac i pa4
cal eval

pr6,   dac 100
    pop
dac pa4
    pop
dac pa3
    jmp x 1
apply, clf 2
jmp apl
"apply"

1kd,
pop
sad . 1
jmp 1k2
push
error 1cd
jmp tfa 2

cn2,
pop
cal cdr get ( & , r ) pair
cond,
sad cn
jmp 1kd
push
cal caar get pi
cal eval
sad cn
jmp cn2
cal car
pop
cal car

eval-2,
cal cdr evaluate result to
eval-1,
cal car
eval,
dac a1
"eval"
sma
jmp ev2 /not atomic
sub cpde
spa
jmp x 1 /number
lac i a1 /atomic symbol
sza
jmp x
error uas
u2,
cal terpri-1
jmp tfa 2
ev2,
lio i a1
cal cdr
dac a2 /argument list
stf 2 come from eval not apply
apl,
lac a1 /function
sma
jmp e3 /non-atomic function
sub cpde /atomic function
sma
jmp e4 /symbol
lac a1 /number
cal cdr
sad (1subr
jmp esu
sas (1fsubr
jmp uaf
lac i a1 /function is fsubr
dap exs
lac a2
dac a1
exs,
lio a2
dac 100
exs,
jmp .
esu, 
  lac 1 a1 /function is subr
  push
  cal evlis-1
  pop
  dap exs
  ral 6s
  and (3
  add (a1
  dac t2
  law a1
  dac t1

sp1,
  sad t2
  jmp sp9
  lac 100
  sad cn
  jmp tfa
  lac 1 100
  dac 1 t1
  lac 100
  cal cdr
  idx t1
  jmp sp1

sp9,
  lac 100
  sas cn
  jmp tma
  lac a1
  jmp exg

e4,
  lac 1 a1 /function is symbol
  sza
  sad a1
  jmp uaf
  dac a1
  jmp apl

uaf,
  error uaf
  jmp u2

e3,
  lac 1 a1 /function is not atomic
  sad (ilambda
  jmp ela
  sad (inlamma
  jmp enl
  sad (illabel
  jmp elb
  lac a2 /evaluate entire function
  push
  lac a1
  cal eval
  pop
  lio 100
  jmp apl-3
ela,     lac a1    /function is "lambda"
push
cal evlis-1
dac a2
pop
dac a1    /args in a2, function in a1
cal cdr    /get lambda variables
/pair lambda list with arg list
el1,     lac a2
sad cn
jmp el9    /no more args
    lac 100
sad cn
jmp tma
lac i 100
lio i a2
bind
idx a2
lac i a2
dac a2
lac 100
cal cdr
jmp el1
el9,     lac 100
sas cn
jmp tfa
lac a1
cal cddr
jmp eval-1
enl,     lac a1    /function is "nlambda"
cal cdr
sad cn
jmp tma
lac i 100
lio a2
bind
idx 100
lac i 100
jmp el9 1
elb,     lac a1    /function is "label"
cal cdr
dac a1
cal cdr
da
lac i a1
bind
jmp apl-1
tfa,     error tfa
stf 4
jmp fal
tma,     error tma
jmp tfa 2
constants
define here x,y
x
y
terminate

define put z
here [define here 123,456 123],[z
456
terminate]
terminate

define pack q
n2=q
n1=767676
repeat 3,n2=n2*100
repeat ifn n2>77,n1=n2^n1\>77^n1
n1
terminate

define pname name,val
pack text1 /name/
pack text2 /name/
1'name=add 
val 1nil
terminate

define su name,num,/g
pname name,add g
put [s name,num,g]
terminate

define fsu name,/g
pname name,add g
put [f name,g]
terminate

define apval name
pname name,1'name
terminate

define thing name
pname name,0
terminate

equals s,if2
equals f,if2
repeat 1-if2,define kill x
repeat if2,define kill x
terminate
equals x,if2

hih, 1

.+\d/
sym,
fsu fix
su revers, 1
fsu trace
tsy,
fsu untrac
thing 99g
thing enter
thing value
esi,
/free storage maker

begin,
        eem
        lio  -1
        dio 0
        szs 20 1
        jmp  5
        lac (and
        dac hih
        law 6301
        jdp asg
        clf 4
        szs 10 1
        jmp nxp

xpl,
    lac (lac-2
    add a2
    dac a1
    cal print
    ty1
    lab
    sas (charac rx
    jmp nxp
    law i 4
    adm a2
    dac snd
    lac i a1
    dap ta5
    sma
    jmp xpl
    lac i ta5
    add (1
    and (-1
    dap low
    jmp xpl

nxp,
    cli
    xct low

gc9,
    sad (frs

ta5,
    law fr2
    dac t1
    dac g1
    idx t1
    dio i t1
    lio g1
    idx t1
    test
    jmp gc9
    dio fre
    jmp 0

constants
sym 2100/
pde,
    pa3, 0
    pa4, 0
    a1, 0
    a2, g3, lac tsy
    a3, g2, 0
eq,
  xor a2
  jmp zerop 1
  /*eq*/

eq4,
  pop
  cal cdr
  lia
  pop
  cal cdr

equal,
  dio t1
  sad t1
  jmp tru
  spaVspi
  jmp eq3
  sma
  spi
  jmp fal
  push
  lai
  push
  lac i 100
  lio i pw1
  cal equal
  sas cn
  jmp eq4
  pop
  /*equal*/

ppf,
  pop
  jmp fal

eq3,
  sub cpde
  swp
  sub cpde
  spaVspi i
  jmp fal
  lac i 100
  xor i t1
  jmp zerop 1

minus,
  cal vag
  jmp crn-1
  /*minus*/

plus,
  cal evl1s
  law cadt2
  /*plus*/

nmop,
  dzm t2
  dap nm2
  lac 100

nm1,
  dac a2
  sad cn
  jmp nm9
  lac i a2
  cal vag

nm2,
  xct
  dac t2

nm3,
  lac a2
  cal cdr
  jmp nm1

nm9,
  lac t2
  jmp crn
cadt2, add t2
times,  cal evlis  //"times"
law 1
dac t2
jsp nmop
jmp . 1
mul t2
scr 1s
dio t2
adm t2
jmp nm3

logand,  cal evlis  //"logand"
clo
dac t2
jsp nmop
and t2

logor,  cal evlis  //"logor"
jsp nmop-1
ior t2

logxor,  cal evlis  //"logxor"
jsp nmop-1
xor t2

greate,  cal vag  //"greaterp"
dac a1
lac a2
cal vag
clo
sub a1
szo
lac 100
jmp atom

remain,  cal divi  //"remainder"
swp
jmp crn

divi,  lai
cal vag
dac a2
lac a1
cal vag
mul c1
div a2
jmp . 2
jmp x
error ovf
jmp nnx

quotie,  cal divi  //"quotient"
jmp crn
and2,    sad cn           /*and*/
          jmp tru
          push
          cal eval-1
          sad cn
          jmp ppf
          pop
          cal cdr
          jmp and2

or1,     pop
          cal cdr

or,      sad cn           /*or*/
          jmp fal
          push
          cal eval-1
          sad cn
          jmp or1

ppt,     pop
          jmp tru

maplis,  sad cn           /*maplist*/
          jmp x
          push
          cal map
          lac i pdl
          dac pwl
          dio i pdl
          jmp mp2

mp1,     push
          cal map
          pop

mp2,     zorch
          sas cn
          jmp mp1
          jmp el5-1

map,     lac a2
          push
          lac i 100
          cal cons-1
          lac a2
          dac a1
          dio a2
          cal apply
          cal cons-1
          pop
          dac a2
          jmp x
append,
  sad cn
  jmp prog2
  push
cal car
cal cons
lac i pd1
dac pw1
dio i pd1
jmp apn2

apn1,
push
cal car
lio a2
cal cons
pop
apn2,
zorch
sas cn
jmp apn1
lio a2
jmp e15

nconc,
  sad cn
  jmp prog2
  dac a2
cal cdr
sas cn
jmp .-3
idx a2
dio i a2
lac a1
jmp x

member,
  lai
  sad cn
  jmp fai
  dac a2
  lac i a2
  lio a1
cal equal
sas cn
jmp x
lac a2
cal cdr
jmp member 1
gensym, law gst  
dac t1  
gen2, idx i t1  
sad (21  
law 1  
dac i t1  
sas (12  
Jmp gen3  
law 20  
dac i t1  
idx t1  
Jmp gen2  
gen3, lac snd  
dac sy2  
dac sy1  
idx sy2  
sub pdl  
add c3  
sma  
Jmp pce  
law charac mg  
ior gst 3  
ral 6s  
ior gst 4  
ral 6s  
dac i sy1  
lac gst  
ral 6s  
ior gst 1  
ral 6s  
ior gst 2  
dac i sy2  
Jmp putob  

constants  
gst, repeat 5,20  

sassoc, lac a2  
sad cn  
Jmp ss2  
cal car  
lac i 100  
sad a1  
Jmp x 1  
lac a2  
cal cdr  
dac a2  
Jmp sassoc 1  

ss2, lio a3  
lac cn  
Jmp ev2 2  

/*gensym*/
prinde, sad cn
jmp pf1
push
cal caar
cal cons-1
lac pq
cal cons
cal cons-1
lac i pd1
cal car
swp
push
swp
cal cons-1
lac pq
cal cons
pop
swp
cal cons
lac (1rplaca
cal cons
cal terpri-1
pop
cal cdr
jmp prinde
pq,
1quote
pf1,
lac (1stop
dac 100
jmp cons-1

constants
dex,
cal cdr
lisa
lac i a1
dac a1
lac lam
cal cons
dio i a1
jmp pn7 2
lam,
1lambda
subst, push
lai
push
cal subs1
pop
pop
jmp x 1

subs1, lio a2
lac a3
cal equal
sad cn
jmp . 3
lac a1
jmp x
lac a3
spa
jmp x
cal cdr
push
lac i a3
dac a3
cal subs1
lio i pdl
dac i pdl
dio a3
cal subs1
lia
pop
dac 100
jmp cons

fix, cal cdr
lio i 100
dio a2
cal cadr
push
cal car
dac a3
lac i a1
dac a1
cal subst
lia
pop
dio i pdl
jmp x

revers, lio cn
sad cn
jmp prog2
push
cal car
cal cons
pop
cal cdr
jmp reverse 1
trace, sad cn
jmp tru
push
lac i pw1
dac t3
lac i t3
sza i
jmp tr2
cal car
sas lam
jmp tr2
lac (199g
cal cons-1
dac t4
lac (1print
cal cons
cal cons-1
lac (1return
cal cons
cal cons-1
lio i pdl
push
lal
cal car
cal cons-1
lac (1value
cal cons
cal cons-1
lac pq
cal cons
cal cons-1
lac (1print
cal cons
lio i pdl
cal cons
dio i pdl
lac i t3
cal cddr
cal car
cal cons-1
lac (199g
cal cons
lac (1setq
cal cons
lio i pdl
cal cons
dio i pdl
lac i t3
cal cadr
lia
lac (1list
cal cons
cal cons-1
lac (1print
lac cons
lio 1 pdl
lac cons
dio 1 pdl
lac t3
cal cons-1
lac (1enter
cal cons
cal cons-1
lac pq
cal cons
cal cons-1
lac (1print
cal cons
lio 1 pdl
cal cons
lac t4
cal cons
lac (1prog
cal cons
lac i t3
cal cddr
dio i 100
idx pdl

tr2,
    pop
cal cdr
    jmp trace

untrac,
    sad cn
    jmp tru
    cal car
    lac i 100
    sza i
    jmp ut2
    cal cddr
dac t2
cal cdar
dac t1
cal caar
sas (199g
    jmp ut2
    lac t1
cal cddr
cal cadr
cal cddr
cal car
dac i t2
    lac a1
    cal cdr
dac a1
    jmp untrac

"untrace"

constants
.+.M1/
frs,
and=and2

equals put,if2
equals pname,if2
equals su,if2
equals fusu,if2
equals apval,if2
equals thing,if2

define s name,num,g
g,jmp ixnum name
isubr
kill g
terminate

define f name,g
g,jmp name
ifsubr
kill g
terminate

here
and=i i
fr2,
equals n1,if2
equals n2,if2
equals n3,if2
equals q,if2
start
(rplaca (quote theor) (quote (lambda (s) (and (null (atom s))
  (th nil nil nil (list s)))))
(rplaca (quote caddr) (quote (lambda (s) (car (caddr s)))))

(rplaca (quote thr) (quote (lambda (a1 a2 c1 c2) (cond ((null a2)
  (and (null (null c2)) (th1 (car c2) a1 a2 c1 (cdr c2))))
  (t (th1 (car a2) a1 (cdr a2) c1 c2))))

(rplaca (quote th1) (quote (lambda (u a1 a2 c1 c2) (cond ((eq (car u) (quote not))
  (th1 (cdr u) a1 a2 c1 c2))
  ((eq (car u) (quote and)) (th2r (cdr u) a1 a2 c1 c2))
  ((eq (car u) (quote or)) (th1 (caddr u) a1 a2 c1 c2))
  ((eq (car u) (quote implie)) (and (th1 (caddr u) a1 a2 c1 c2)
  (th1 (caddr u) a1 a2 c1 c2))
  ((eq (car u) (quote equiv)) (and (th1 (caddr u) (caddr u) a1 a2 c1 c2)
  (th2r (cdr u) a1 a2 c1 c2))))

(rplaca (quote th2r) (quote (lambda (u a1 a2 c1 c2) (cond ((eq (car u) (quote not))
  (th1 (caddr u) a1 a2 c1 c2))
  ((eq (car u) (quote and)) (th2r (cdr u) a1 a2 c1 c2))
  ((eq (car u) (quote or)) (th2r (caddr u) a1 a2 c1 c2))
  ((eq (car u) (quote implie)) (th1 (caddr u) (caddr u) a1 a2 c1 c2))
  ((eq (car u) (quote equiv)) (and (th1 (caddr u) (caddr u) a1 a2 c1 c2)
  (th2r (cdr u) a1 a2 c1 c2))))

(rplaca (quote th11) (quote (lambda (u v1 v2 a1 a2 c1 c2) (cond ((atom u)
  (or (member v1 c1) (th (cons v1 a1) a2 c1 c2)))
  (t (or (member v1 v2) (th a1 (cons v1 v2) a2 c1 c2))))

(rplaca (quote th1r) (quote (lambda (u v1 v2 a1 a2 c1 c2) (cond ((atom u)
  (or (member v1 a1) (th a1 a2 (cons v1 c1) c2)))
  (t (or (member v1 v2) (th a1 a2 c1 (cons v1 c2))))

(rplaca (quote th21) (quote (lambda (v1 v2 a1 a2 c1 c2) (cond ((atom v1)
  (or (member v1 a1) (th1 (car v) c1) (th1 (cadr v) (cons (car v) a1) a2 c1 c2)))
  (t (or (member v1 v2) (th1 (car v) a2 (cons (car v) c1))))

(rplaca (quote th2r) (quote (lambda (v1 v2 a1 a2 c1 c2) (cond ((atom v1)
  (or (member v1 a1) (th1 (cadr v) a1 a2 (cons (car v) c1) c2))
  (t (or (member v1 v2) (th1 (cadr v) a2 c1 c2))))

(rplaca (quote th11) (quote (lambda (v1 v2 a1 a2 c1 c2) (cond (
  (atom v1) (or (member v1 c1) (th1 v2 (cons v1 a1) a2 c1 c2)))
  (t (or (member v1 v2) (th1 v2 a1 (cons v1 v2) a2 c1 c2)))))))

(stop)
(rplaca (quote thing) (quote (equiv (and (and (equiv p q) (or s (not r))) (equiv r q)) (or (or (and (and p q) (and r s)) (and (not p) (not q)) (and (not r) (not s))) (and (and (not p) (not q)) (and (not r) s)))))))

(stop)